Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Aljaž Košir

Horizontalno skaliranje spletnih aplikacij

diplomsko delo

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Boštjan Slivnik

Ljubljana, 2017

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Horizontalno skaliranje spletnih aplikacij predstavlja enega od večih različnih načinov zagotavljanja smotrne uporabe računalniških virov v sistemu. Preučite različna orodja, ki omogočajo izvajanje spletnih aplikacij v kontejnerjih, in izdelajte prototipno okolje, ki bo omogočalo avtomatično horizontalno skaliranje teh spletnih aplikacij. Poleg tega naj okolje omogoča merjenje porabe virov in sprotni nadzor preko spletnega vmesnika. Ustrezna orodja in programske pakete izberite sami. Delovanje izdelanega orodja ustrezno preverite z meritvami, ki dokazujejo uporabo horizontalnega skaliranja.

Zahvaljujem se mentorju Boštjanu Slivniku, ki mi je nudil strokovno pomoč pri razvijanju diplomske naloge in prav tako pri pisanju diplomske naloge. Zahvaljujem se Gergoru Pipanu in podjetju XLAB, ki sta mi omogočila razvoj diplomskega dela na njihovi infrastrukturi. Zahvaljujem se tudi Alešu Černivcu, ki je nudil pomoč pri razvijanju teme diplomske naloge. Posebna zahvala gre tudi družini, ki me je ves čas podpirala pri izobraževanju.

Kazalo

Povzetek

Naslov: Horizontalno skaliranje spletnih aplikacij

Avtor: Aljaž Košir

V diplomski nalogi je predstavljeno okolje za avtomatsko horizontalno skaliranje, ki se glede na metrike, ki jih okolje izvaža v podatkovno bazo, odzove z ustvarjanjem ali brisanjem kontejnerjev ali navideznih računalnikov. Okolje poleg skaliranja tudi izriše grafe o porabi posameznih virov v našem sistemu (poraba pomnilnika, poraba procesorja, število poslanih paketov preko omrežja ...), na katerih lahko spremljamo delovanje sistema. Ustvarjeno okolje temelji na odprtokodnih orodjih OpenStack, Docker, Grafana, Prometheus, Alertmanager, cAdvisor, Node exporter, Vagrant in Ansible.

Ključne besede: skaliranje, Docker, horizontalno skaliranje.

Abstract

Title: Horizontal scaling of Web applications

Author: Aljaž Košir

This thesis is devoted to design environment for automatic horizontal scaling, which will, based on the metrics, that are provided by our environment and saved into database, respond with creation or removal of virtual machines. Environment also has graphs representing resource usages (memory usage, processor usage, numbers of network packets sent, ...), on which we can monitor our system. Environment is built with the help of open source tools OpenStack, Docker, Grafana, Prometheus, Alertmanager, cAdvisor, Node exporter, Vagrant and Ansible.

Keywords: scaling, Docker, horizontal sacling.

Poglavje 1

Uvod

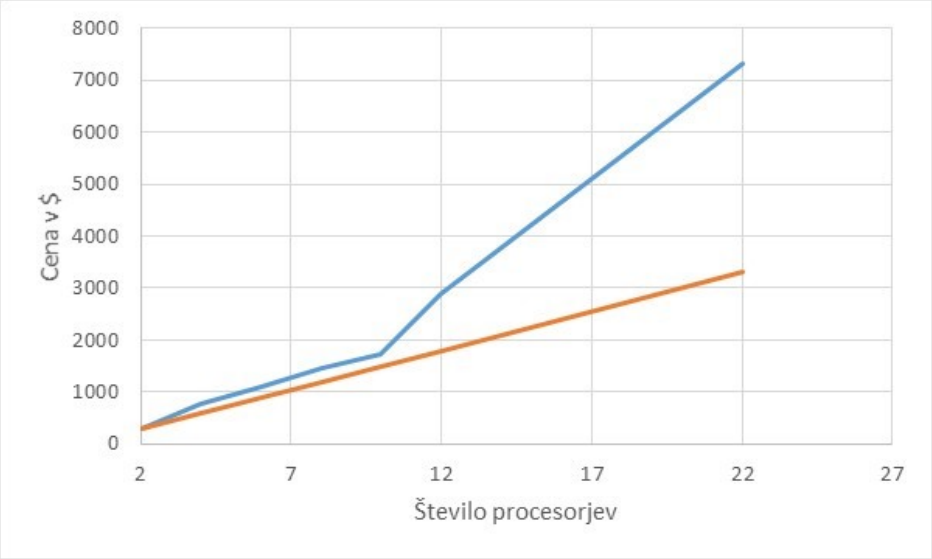
Z vse večjo uporabo spletnih aplikacij v vsakdanjem življenju se pojavi potreba po visoki odzivnosti aplikacije. Eden izmed načinov za zagotavljanje odzivnosti je skaliranje (angl. *scaling*) spletne aplikacije. Ko govorimo o skaliranju, mislimo na razširitev strojne opreme v sistemu, s katerim izboljšamo odzivnost aplikacije. Poznamo dve glavni vrsti skaliranja, vertikalno in horizontalno.

Vertikalno skaliranje se začne zelo preprosto. Predpostavimo, da imamo delujočo spletno stran, postavljeno na strežniku. Ko strežnik ni več dovolj zmogljiv zaradi prevelikega obiska spletne strani, ga zamenjamo z močnejšim. Če se v prihodnosti spet pojavi potreba po močnejšem strežniku, ga preprosto zamenjamo z boljšim. Problem, ki nastane ob uporabi vertikalnega skaliranje, je nedostopnost aplikacije med nadgraditvijo strežnika. Problem seveda lahko ublažimo tako, da preden nadgradimo strežnik, ustvarimo kopijo strežnika in vse uporabnike preusmerimo nanjo.

Horizontalno skaliranje se od vertikalnega razlikuje v tem, da pri horizontalnem ne zamenjamo celotnega strežnika, ampak dodamo nov strežnik že obstoječemu. Problem tega je velika količina strojne opreme, kar pomeni več vzdrževanja. Paziti moramo tudi pri nakupu strežnika, saj želimo tak strežnik, da bodo vsi viri približno enako uporabljeni. Tako ne pridemo do problema, kjer imamo 10 strežnikov, na katerih je na primer poraba pomnil-

nika zelo visoka, poraba procesorja pa zelo nizka.

Strojna oprema je pri horizontalnem modelu sicer cenejša, ker imamo več manjših sistemov, vendar je zaradi tega dražje vzdrževanje. Na sliki 1.1 vidimo primerjavo cen nakupa strojne opreme med horizontalnim in vertikalnim skaliranjem. Večina ponudnikov skaliranja ne zaračunava in vsi ponujajo



Slika 1.1: Primerjava cen strojne opreme pri horizontalnem oziroma vertikalnem skaliranju

vertikalno ter tudi horizontalno skaliranje. Predstavitev ponudnikov in primerjave cen si bomo pogledali v naslednjem poglavju.

V diplomski nalogi smo uporabili horizontalno skaliranje z uporabo virtualizacije. To pomeni, da velik strežnik razdelimo na več manjših enot. Poznamo več programskih rešitev, ki se ukvarjajo z virtualizacijo (VirtualBox, Docker ...). V naslednjih poglavjih si bomo ogledali, kako smo skalirali sistem za horizontalno skaliranje aplikacije s pomočjo orodja, ki omogoča zagon aplikacije v kontejnerjih, in kako smo merili metrike, na katere se sistem odziva.

Poglavje 2

Pregled področja

V tem poglavju predstavimo različne ponudnike, kakšno skaliranje uporabljajo in primerjamo cene. Predstavimo tudi različne tehnologije za virtualizacijo. Na koncu poglavja pa si še ogledamo, zakaj smo za diplomsko nalogo izbrali tehnologijo Docker.

2.1 Virtualizacija

Virtualizacija je postopek, pri katerem ustvarimo navidezni vir. Navidezni vir je lahko računalniška strojna oprema, datotečni sistem, omrežni vir in podobno. Čeprav nekateri načini virtualizacije negativno vplivajo na zmogljivost, nam tehnologije rešujejo problem prenosljivosti aplikacij. Z virtualizacijo lahko zagotovimo enake pogoje za izvajanje naše aplikacije na različni strojni opremi ali različnem operacijskem sistemu. Pogledali si bomo dva glavna načina virtualizacije, to sta virtualizacija na nivoju strojne opreme in virtualizacija na nivoju operacijskega sistema.

2.2 Virtualizacija na nivoju strojne opreme

Virtualizacija na nivoju strojne opreme je postopek, pri katerem upravljavec navideznih naprav oziroma hipervisor ustvari navidezni računalnik, ki se

obnaša kot pravi računalnik z operacijskim sistemom. Programska oprema, ki se izvaja na teh virtualnih napravah, je ločena od osnovnih virov strojne opreme. Hipervisor ustvari virtualno strojno opremo, torej je programska oprema odvisna od virtualne strojne opreme. Vsaka virtualna naprava je izolirana od drugih. Če se ena izmed virtualnih naprav okuži z zlonamerno programsko opreme, to ne bo vplivalo na delovanje ostalih virtualnih naprav.

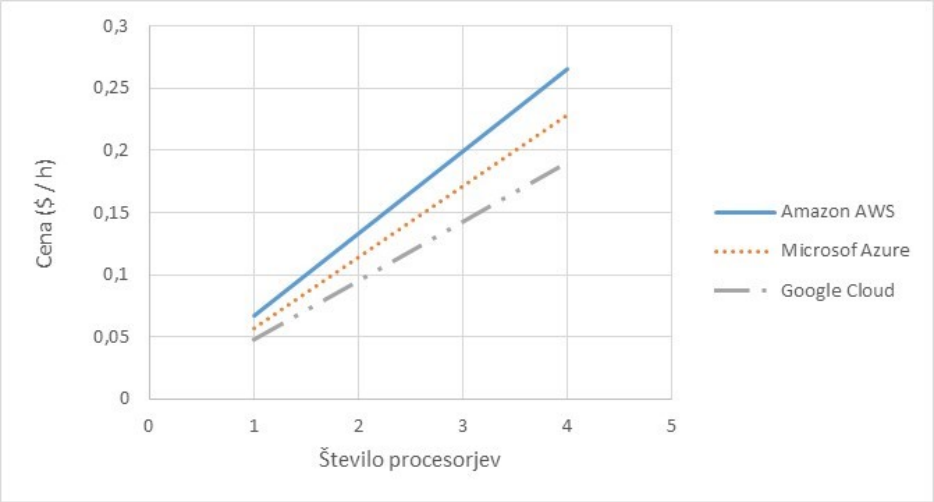
Virtualizacijo strojne opreme lahko izvajamo tudi na osebnem računalniku z uporabo različnih programskih orodij (VirtualBox, VMware ...). Če naš računalnik ni dovolj zmogljiv ali pa želimo virtualno napravo najeti, lahko uporabimo enega izmed ponudnikov teh storitev.

Pogledali si bomo glavne ponudnike, Amazon [3], Google [8] in Microsoft [10]. Vsi trije ponudniki ponujajo horizontalno in tudi vertikalno skaliarnje. Avtomatsko skaliranje je možno samo za horizontalno skaliranje, če pa želimo vertikalno skaliranje, moramo za to poskrbeti sami. Skaliranje je pri ponudnikih brezplačno oziroma je že všteto v ceno zakupa navideznih računalnikov. Cena pri ponudnikih se zaračunava tako, da plačamo toliko, kolikor porabimo, če je naš strežnik v mirovanju, ne plačujemo nič. Cena se zaračunava v enoti $/h, primerjavo cen med ponudniki si lahko pogledamo v tabeli 2.1. Poraba diska se zaračunava glede na število uporabljenih gigabajtov, predpostavljamo, da na mesec porabimo 50 GB diska, kar nas stane dodatnih $0.00005475 na mesec. Zaradi kompetitivnosti med ponudniki se cene in načini plačevanje spreminjajo zelo hitro. Pri horizontalnem skaliranju moramo paziti, saj je nekaj programske opreme plačljiva in se hkrati lahko uporablja samo na enem računalniku.

Iz tabele 2.1 primerjamo cene med različnimi ponudniki, s slike 2.1 pa je razvidno, da bi cena pri vertikalnem skaliranju naraščala linearno, tako kot pri horizontalnem. V primeru vertikalnega skaliranje moramo za skaliranje skrbeti sami, za horizontalno skaliranje pa za nas poskrbi ponudnik.

	Amazon AWS	Microsoft Azure	Google Cloud
1CPU, 3.75GB RAM	\$0.067 / h	\$0.057 / h	\$0.0475 / h
2CPU, 7.5GB RAM	\$0.133 / h	\$0.114 / h	\$0.095 / h
4CPU, 15GB RAM	\$0.266 / h	\$0.229 / h	\$0.190 / h

Tabela 2.1: Primerjava cen med ponudniki.



Slika 2.1: Primerjava cen med ponudniki.

2.3 Virtualizacija na nivoju operacijskega sistema

Virtualizacija na nivoju operacijskega sistema je virtualizacija, kjer jedro operacijskega sistema dovoljuje obstoj več izoliranih primerkov uporabniškega prostora. Ponavadi jim rečemo kontejnerji (angl. *containers*) in so videti kot pravi računalniki iz pogleda programa, ki uporablja njihove vire. Kontejnerji so prav tako kot pri virtualizaciji strojne opreme izolirani med seboj. Ena izmed slabosti pri uporabi opisane virtualizacije je fleksibilnost. Zaradi odvisnosti od jedra operacijskega sistema pomeni, da istega kontejnerja ne moremo zagnati na operacijskem sistemu Linux in operacijskem sistemu Windows. Prednost pred virtualizacijo na nivoju strojne opreme (ustvarja-

nje navideznega računalnika) je hitrost ustvarjanja in zagona, saj trajata le nekaj sekund, pri ustvarjanju navideznega računalnika pa več minut.

To virtualizacijo lahko prav tako kot virtualizacijo strojne opreme izvajamo doma na osebnem računalniku. Če pa želimo, lahko navidezni računalnik najamemo pri enem izmed prej opisanih ponudnikov in nanj namestimo omenjeno programsko opremo za virtualizacijo. Poznamo več različnih programskih rešitev za izvajanje virtualizacije na nivoju operacijskega sistema. Med najbolj znane sodijo naslednja orodja:

- **Docker** izolira proces in ga ovije v datotečni sistem, ki vsebuje vse potrebno za izvajanje procesa [6]. Procesu tako zagotavlja nemoteno izvajanje ne glede na operacijski sistem. Docker uporablja aplikacijske kontejnerje, kar pomeni, da je namenjen izvajanju enega procesa, ker pa so procesi ločeni, naredi našo aplikacijo zelo lahko skalabilno. Docker težavo o fleksibilnosti, ki preprečuje izvajanje kontejnerja na različnih jedrih, rešuje z virtualizacijo operacijskega sistema Linux, kar negativno vpliva na zmogljivost. Docker si vire deli z gostujočim operacijskim sistemom, torej če jih imamo na voljo 2 GB, jih ima na voljo tudi Docker.

- **Rkt** je bil razvit kot bolj varna alternative Dockerju. Starejše verzije Dockerja so zahtevale zagon kontejnerjev kot root. Glavna izvršitvena enota pri rkt je kapsula (angl. *pod*), skupek ene ali več aplikacij, ki se izvajajo v skupnem konteksu [13]. Rkt za razliko od Dockerja uporablja standardno specifikacijo za ustvarjanje slik, ki jo imenujejo specifikacija aplikacijskih konetjnerjev. Omogoča tudi zagon drugih kontejnerskih slik, kot na primer tiste, ustvarjenje z Dockerjem. Vsaka kapsula se izvaja v svojem izoliranem okolju.

- **OpenVZ** vidi kontejner kot navidezni računalnik, v njem lahko poganjamo več aplikacij za razliko od prej opisanih orodij. Vsak kontejner deluje kot samostojen fizični strežnik. Ponuja virtualizacijo, kjer imamo

lahko hkrati več operacijskih sistemov na enem računalniku. Zaradi odvisnosti od jedra pa lahko poganjamo samo operacijski sistem Linux.

Za Docker smo se odločili zaradi velike skupnosti, ki jo ima Docker, tako smo lahko hitro dobil odgovor, če smo naleteli na kak problem. Razlog je bil tudi, da smo imeli v podjetju, kjer opravljam študentsko delo, veliko ljudi, ki so že uporabljali Docker in so lahko nudili pomoč.

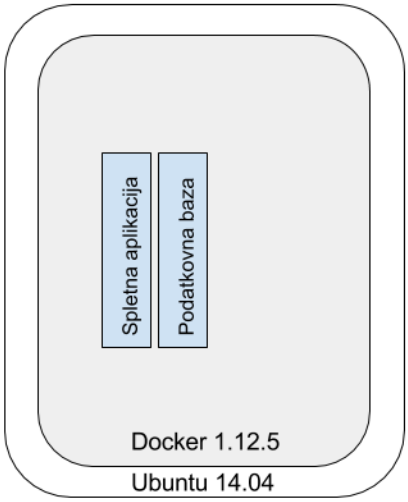
Poglavje 3

Arhitektura sistema za horizontalno skaliranje spletne aplikacije

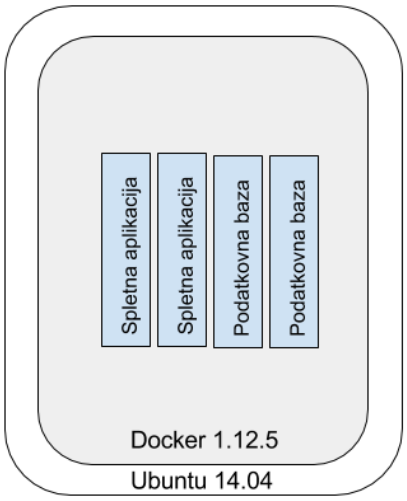
Želimo postaviti sistem, ki bo horizontalno skaliral spletno aplikacijo glede na porabo virov, ki jih imamo na voljo. Spremljanje porabe in odločanje o skaliranju opišemo v podpoglavjih, kjer opisujemo podporna orodja. Na začetku, ko se naš sistem postavi, dobimo arhitekturo, ki si jo lahko ogledamo na sliki 3.1. Imamo neki operacijski sistem, znotraj operacijskega sistema imamo programsko rešitev za virtualizacijo na nivoju operacijskega sistema. Na slikah je predpostavljeno, da za operacijski sistem uporabljamo Linux, za virtualizacijo na nivoju operacijskega sistema pa uporabljamo Docker. Orodje za virtualizacijo na nivoju operacijskega sistema na začetku upravlja samo dva kontejnerja, prvi je kontejner, v katerem je zagnana spletna aplikacija, drugi pa je kontejner, kjer imamo podatkovno bazo za to aplikacijo.

Ko kontejnerja nista več zmožna vzdrževati zadostnega števila uporabnikov, dodamo še nove kontejnerje. Tako dobimo stanje, ki ga vidimo na sliki 3.2. Podvojili smo kontejnerje za spletno aplikacijo in za podatkovno bazo.

Aplikacija je zagnana na enem navideznem računalniku. Če bo našo



Slika 3.1: Začetna arhitektura.



Slika 3.2: Arhitektura po ustvarjanju novih kontejnerjev.

aplikacijo začelo uporabljati še več uporabnikov, bo našemu navideznemu računalniku zmanjkalo virov. Ker sistem skaliramo horizontalno, bomo že obstoječemu navideznemu računalniku dodali še en navidezni računalnik. Zadnji korak vidimo na sliki 3.3. Imamo nov navidezni računalnik, na katerem lahko ponovno skaliramo kontejnerje. Ko zmanjka virov, pa ponovno dodamo nov navidezni računalnik.

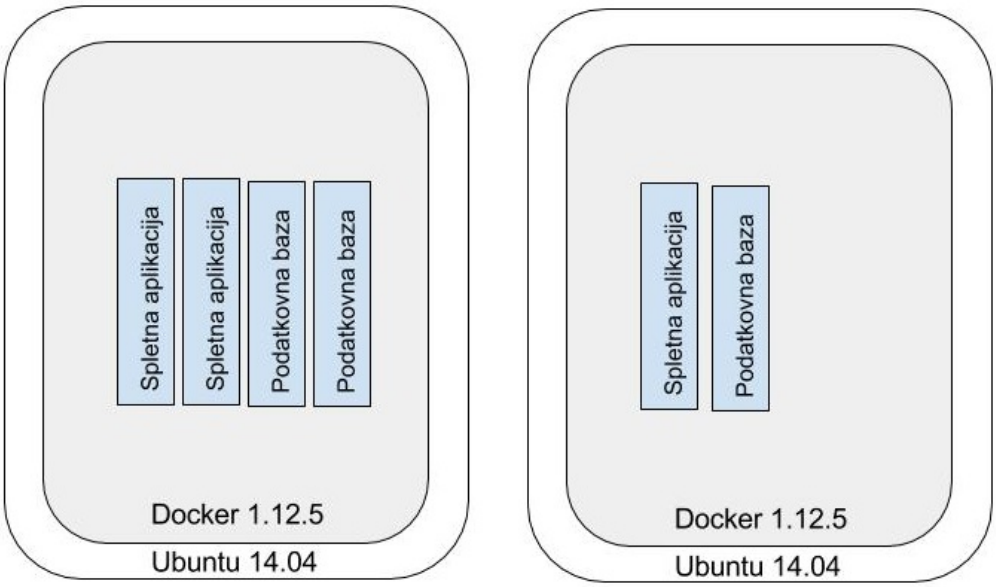
Celoten sistem mora torej znati opravljati naslednje operacije:

1. ustvarjanje prvotnega navideznega računalnika,

2. namestitev programske opreme za delovanja sistema za horizontalno skaliranje,

3. ustvarjanje in brisanje navideznih računalnikov ali kontejnerjev ter

4. izris metrik.



Slika 3.3: Arhitektura po ustvarjanju novega navideznega računalnika.

3.1 Ustvarjanje navideznega računalnika

Prvotni navidezni računalnik je ustvarjen s pomočjo orodja Vagrant 1.9.5. Vagrant je orodje za ustvarjanje in upravljane navideznih računalnikov [14]. Vagrant se uporablja za ustvarjanje navideznih računalnikov na lokalnem računalniku, lahko pa z uporabo modulov ustvarimo tudi navidezne računalnike v oblaku. Ker so naši navidezni računalniki ustvarjeni na strežniku Openstack, smo uporabili modul vagrant-openstack-provider. Openstack je zbirka odprtokodnih programskih projektov, ki omogočajo postavitev in poganjanje infrastrukture za računstvo v oblaku. Za postavitev navideznega računalnika Vagrant potrebuje skripto, ki jo lahko vidimo na izseku izvorne kode 3.1. Najbolj pomembne nastavitve pri ustvarjanju navideznega računalnika so naslednje:

1. zagonska slika navideznega računalnika (angl. *image*), ki vsebuje navidezni disk, na katerem je naložen zagonski operacijski sistem,

2. okus (angl. *flavor*), ki opredeljuje število navideznih procesorjev, velikost pomnilnika in velikosti navideznega diska,

3. varnostne skupine (angl. *security groups*), ki opredeljujejo, katera vrata (angl. *ports*) so na navideznih računalnikih odprta,

4. ime omrežja (angl. *networks*), ki nam omogoča komunikacijo med navideznimi računalniki v istem omrežju in dostop do spleta,

5. imena skript za zagon (angl. *provisioning*), ki se poženejo ob zagonu navideznega računalnika,

6. ime SSH-ključa (angl. *keypair name*), ki določa, kateri ključ bo uporabljen za avtentikacijo na navidezni računalnik.

Nastavitve, ki jih opisujemo, so samo imena prednastavljenih nastavitev. Nastavitev okusa m1.small na primer predstavljajo 1 navidezni procesor, 2 GB pomnilnika in 10 GB prostora na disku.

Zagonska slika navideznega računalnika je ustvarjena preko vmesnika OpenStack. Nanjo je naložen operacijski sistem Ubuntu 14.04. Na zagonsko sliko so naloženi Docker 1.12.5 in vse knjižnice za njegovo delovanje. Zagonska slika je prenesena na isti strežnik kot Openstack. S tem zagotovimo, da imajo vsi navidezni računalniki naloženo enako programsko opremo, hkrati pa se pohitri tudi zagon navideznega računalnika, saj nam ni treba po vsakem zagonu namestiti še vse potrebne programske opreme. Prvotni navidezni računalnik ima na voljo 4 GB pomnilnika in 2 navidezni procesorski enoti.

3.2 Namestitev programske opreme za delovanje sistema za horizontalno skaliranje

Namestitev programske opreme za delovanja sistema za horizontalno skaliranje je narejeno s pomočjo orodja Ansible 2.3.1. Ansible je orodje za avtomatsko konfiguracijo navideznega računalnika [4]. Po tem, ko imamo

```
Vagrant.configure(2) do |config|
  config.vm.define "docker-manager-openstack" do |
    ↪ manager|
    manager.vm.provider :openstack do |os, override|
      os.openstack_auth_url = 'http
        ↪ ://10.10.43.2:5000/v3'
      os.username = 'aljaz_kosir'
      os.password = 'xxxxxxxx'
      os.domain_name = 'xlab'
      os.project_name = 'aljaz-kosir'
      os.identity_api_version = '3'
      os.image = 'docker-ubuntu'
      os.flavor = 'm1.small'
      os.security_groups = ['default', 'diplomska']
      os.networks = ['aljaz-kosir']
      os.floating_ip_pool = 'xlab'
      os.keypair_name = 'Aljaz'
    end
    manager.vm.provision "ansible" do |ansible|
      ansible.groups = {
        "managers" => ["docker-manager"]
      }
      ansible.playbook = "provisioning/init-swarm.
        ↪ yml"
    end
  end
end
```

Izsek izvorne kode 3.1: Vagrantfile

postavljen delujoč navidezni računalnik z operacijskim sistemom, z orodjem Ansible skonfiguriramo in namestimo želeno programsko opremo. Ansible skripte, ki se jim reče playbook, so napisane v preprostem programskem jeziku YAML [15].

3.2.1 Namestitev orodja Docker Swarm

Docker Swarm je orodje znotraj Dockerja, ki omogoča ustvarjanje gruče kontejnerjev Docker [7]. Kontejnerji tečejo na enem ali več virtualnih računalnikov, skupaj pa tvorijo enotno, izolirano omrežje, ki ga imenujemo swarm. Z orodjem lahko ustvarjamo ali brišemo storitve, ki so zagnane v kontejnerjih Docker. Storitve Docker Swarm so kot storitve v operacijskem sistemu, le da delujejo neodvisno od operacijskega sistema. Docker Swarm poskrbi, da so naše storitve vedno dosegljive. Tudi v primeru, če se naša storitev nepredvidljivo ugasne, jo Docker Swarm ponovno zažene.

Ko se navidezni računalnik vzpostavi, orodje Ansible inicializira Docker Swarm. Inicializiramo ga z ukazom:

```
docker swarm init --advertise--addr=<IP naslov
  ↪ streznika>.
```

Ukaz kot izhod vrne žeton, ki ga bodo kasneje vsi ostali navidezni računalniki uporabili za pristop v swarm. Žeton in IP-naslov se shranita v datoteko za kasnejšo uporabo.

3.2.2 Namestitev spletne aplikacije

Spletna aplikacija, ki smo jo uporabili kot primer za horizontalno skaliranje, je bila razvita v podjetju XLAB, in sicer v okvirju evropskega projekta ACDC [1], glavni namen projekta pa je bil boj proti kibernetskim napadom. Gre za trinivojsko spletno aplikacijo, ki je napisana v programskem jeziku Java. Aplikacijo je bilo treba sprva spraviti v kontejner Docker, da smo jo lahko uporabil kot storitev Docker v Docker Swarmu. Kontejner Docker se ustvari s pomočjo zagonske slike Docker, ki jo ustvarimo s pomočjo datoteke, imenovane Dockerfile. Primer izvorne kode zagonske slike lahko vidimo na

```
FROM ubuntu:12.04

RUN apt-get update && apt-get install -y \
ant \
openjdk-7-jre \
openjdk-7-jdk \
tomcat7 \
curl

RUN apt-get -y remove openjdk-6-jre openjdk-6-jre-
   ↪ lib

RUN apt-get -y install mysql-client

COPY GCMServer /opt/GCMServer
WORKDIR /opt/GCMServer

EXPOSE 8080

RUN /opt/GCMServer/install.sh
HEALTHCHECK --interval=10s --timeout=3s --retries=50
   ↪  CMD curl -f http://localhost:8080/GCMServer/
   ↪ login || exit 1
CMD ["/opt/GCMServer/entrypoint.sh"]
```

Izsek izvorne kode 3.2: Dockerfile

izseku izvorne kode 3.2. Datoteka je skupek konfiguracijskih ukazov, ki skonfigurirajo in ustvarijo zagonsko sliko Docker. V datoteko Dockerfile najprej vnesemo ukaze, ki namestijo knjižnice za povezovanje do podatkovne baze MariaDB, ki bo prav tako ustvarjena kot storitev Docker, nato namestimo še Apache Tomcat, ki streže našo aplikacijo. Na koncu Dockerfile z ukazom `CMD` povemo še, katero datoteko mora Docker uporabiti za zagon storitve. V diplomski nalogi je za primer uporabljena spletna aplikacija ACDC, lahko pa bi uporabili katero koli drugo aplikacijo.

S pomočjo že obstoječe zagonske slike Docker (mariadb-cluster) ustvarimo podatkovno bazo mariadb. Ko se baza uspešno vzpostavi, s pomočjo poizvedenega jezika SQL vstavimo še začetne podatke, kot so uporabnik in nekaj testnih podatkov.

3.2.3 Namestitev orodja za izvoz metrik

Po namestitvi aplikacije namestimo orodja za izvoz metrik, in sicer Node exporter ter cAdvisor. Metrike obeh orodij lahko shranjujemo v poljubno podatkovno bazo, ki omogoča shranjevanje glede na čas. V našem primeru je uporabljena časovna podatkovna baza iz sistema Prometheus.

Node exporter

Node exporter je orodje za izvoz metrik o strojni opremi in operacijskem sistemu [11]. Za razliko od cAdvisorja Node exporter izvaža metrike glede na celoten navidezni računalnik. Prav tako kot pri podatkovni bazi MariaDB uporabimo že obstoječo zagonsko sliko Docker. Zagonska slika nam omogoča zagon Node exporterja kot storitev Docker. Nastaviti je treba, katere metrike bomo nadzorovali. Za naše potrebe vključimo naslednje metrike:

1. `diskstats` izpostavlja vhodne/izhodne statistike diskov,
2. `meminfo` izpostavlja statistike pomnilnika,
3. `netstat` izpostavlja omrežne statistike,
4. `stat` izpostavlja statistike procesorja na sistemu in porabo procesorja.

cAdvisor

cAdvisor je orodje, ki izvaža podatke o uporabi virov v kontejnerjih [5]. Je proces, ki zbira, agregira, procesira in izvozi podatke. Orodje cAdvisor za razliko od Node exporterja nima nobenih dodatnih nastavitev, dobimo pa že vgrajen spletni vmesnik za nadzor nad kontejnerji.

3.2.4 Namestitev sistema Prometheus in orodja Alertmanager

Prometheus je odprtokodni sistem za nadzorovanje, opozarjanje in hranjenje podatkov [12]. Ima multidimenzionalno podatkovno bazo za shranjevanje podatkov glede na čas. Za vpogled v bazo je uporabljen poizvedovalni jezik, razvit s strani organizacije Prometheus. Sistem nam ponuja upravljavca za alarme, to je orodje, ki nas opozarja o prekoračenih metrikah, imenujemo ga Alertmanager [2]. Sistem vsebuje tudi časovno podatkovno bazo, ki jo zaženemo kot kontejner Docker s pomočjo že obstoječe zagonske slike Docker.

Organizacija Prometheus ima tudi zagonsko sliko za upravljavca alarmov Docker, vendar jo je bilo treba za potrebe diplomske naloge prilagoditi. Dodali smo nastavitve, da smo bili o alarmih obveščeni preko aplikacije Slack in na upravljavca navideznih računalnikov, o katerem bomo več razložili kasneje. Prav tako smo upravljavca skonfigurirali tako, da ima vsak alarm svojo skupino, kar bo pomenilo, če se alarm proži večkrat hkrati, bo upravljavec alarmov obvestilo poslal samo enkrat. Tak primer nastane, če vsem našim navideznim računalnikom primanjkuje procesorske moči in vsi prožijo alarm.Treba je bilo tudi definirati alarme. Primer definicije alarma lahko vidimo na izseku izvorne kode 3.3. V primeru na izseku izvorne kode se alarm proži, ko metrika `process_cpu_seconds_total` preseže vrednost 70% in je vrednost presežena za 5 minut. Meje metrik bomo opisali v naslednjem poglavju.

3.2.5 Namestitev upravljavca navideznih računalnikov

Upravljavec navideznih računalnikov je program python, ki se glede na dobljene alarme odzove, ali bo ustvaril ali brisal navidezni računalnik. Prav tako kot vsi drugi programi je tudi ta ustvarjen kot Docker container. Podrobnejši opis in delovanje sledita v naslednjem podpoglavju.

```
ALERT high_cpu_usage_on_node
  IF sum(rate(process_cpu_seconds_total[5m])) by (
    ↪ instance) * 100 > 70
  FOR 5m
  ANNOTATIONS {
      summary = "HIGH CPU USAGE WARNING ON '{{
        ↪ $labels.instance }}'",
      description = "{{ $labels.instance }} is using
        ↪  a LOT of CPU. CPU usage is {{ humanize
        ↪ $value}}%.",
  }
```

Izsek izvorne kode 3.3: Definicija alarma

3.3 Ustvarjanje in brisanje navideznih računalnikov ali kontejnerjev

Ustvarjanje in brisanje navideznih računalnikov poteka preko upravljavca navideznih računalnikov. Upravljavec je spletna storitev, narejena s pomočjo pythona in knjižnice Flask. Deluje kot REST-storitev (angl. *Representational State Transfer*). Ima dve končni točki (angl. *endpoint*), eno za brisanje in drugo za ustvarjanje navideznih računalnikov. Vsak navidezni računalnik ima maksimalno tri spletne aplikacije in eno podatkovno bazo. Če se hkrati pošlje več istih zahtevkov iz različnih navideznih računalnikov, se upošteva sam eden, saj so alarmi z istim imenom v isti skupini, kot smo že omenili v podpoglavju 3.2.4.

3.3.1 Ustvarjanje navideznih računalnikov

Ustvarjanje navideznih računalnikov se začne z zahtevkom POST na končno točko našega upravljavca za alarme `/servers/` s strani upravljavca za alarme. Upravljavec nato ustvari nove kontejnerje za našo aplikacijo, če imamo na

voljo proste vire. Če prostih virov ni, upravljavec navideznih računalnikov s pomočjo modula `novaclient` ustvari navidezni računalnik na strežniku in ga poimenuje `docker-node-{zaporedna_stevilka}`. V podpoglavju 3.2.1 smo omenili shranjevanje žetona in IP-naslova v datoteko. Iz te datoteke zdaj preberemo žeton in IP-naslov, ki ju uporabimo za pridružitev v omrežje swarm. Izvede se ukaz

```
docker swarm join <zeton> <IP naslov streznika>
```

Ko je navidezni računalnik uspešno vključen v omrežje swarm, na njem z ukazoma

```
docker scale acdc=<stevilo spletnih aplikacij + 3>
```

in

```
docker swarm scale db=<stevilo podatkovnih baz + 1>
```

ustvarimo tri spletne aplikacije in eno podatkovno bazo. To je maksimalno število spletnih aplikacij oziroma podatkovnih baz, ki jih lahko ustvarimo na navideznem računalniku z viri, ki jih imamo na voljo. Navidezni računalniki imajo v našem primeru na voljo 2 GB pomnilnika in eno navidezno procesorsko enoto, lahko pa bi omejitve nastavili tudi drugače.

3.3.2 Brisanje navideznih računalnikov

Brisanje navideznih računalnikov se začne z zahtevkom DELETE na končno točko našega upravljavca za alarme `/servers/` s strani upravljavca za alarme. Upravljavec med navideznimi računalniki, ki so na voljo, izbere tistega z najvišjo zaporedno številko in na njem izvede ukaz

```
docker swarm leave
```

To pomeni, da bo navidezni računalnik zapustil omrežje Docker Swarm. Docker Swarm bo zaznal, da zaradi neodzivnega navideznega računalnika primanjkujejo tri spletne aplikacije in ena podatkovna baza ter bo poskušal ustvariti manjkajoče storitve na že obstoječih navideznih računalnikih. Upravljavec z ukazom

```
docker scale acdc=<stevilo spletnih aplikacij - 3>
```

in

```
docker swarm scale db=<stevilo podatkovnih baz - 1>
```

prepreči poskus ustvarjanja manjkajočih storitev. Na koncu še s pomočjo modula `novaclient` izbrišemo neuporabljen navidezni računalnik.

3.4 Izris metrik

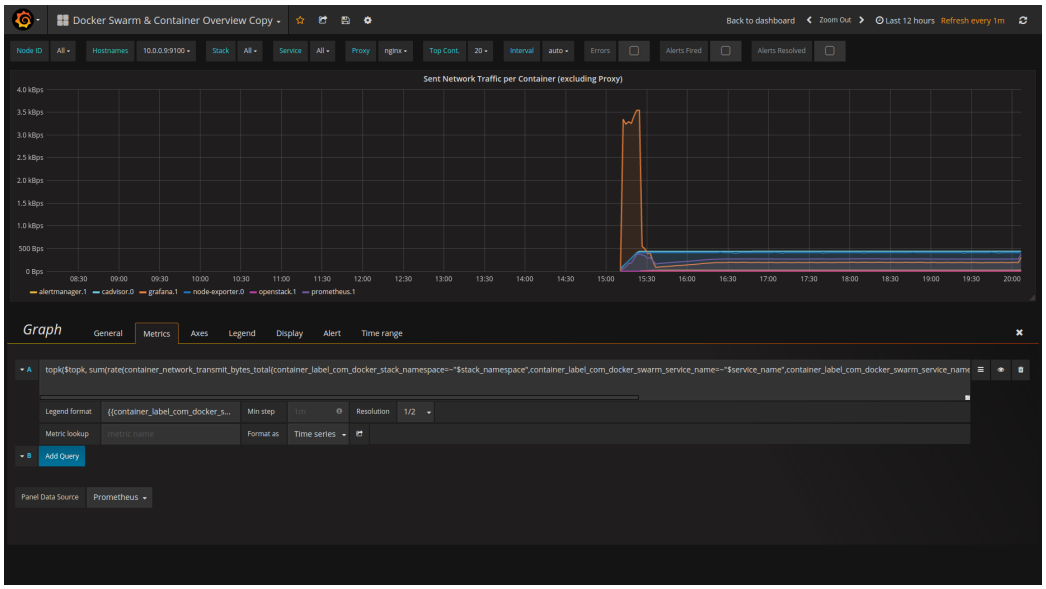
Izrisovanje metrik smo izvedeli z uporabo orodja Grafana [9]. Grafana je prav tako kot vsi drugi programi zagnan kot kontejner Docker. Je orodje, preko katerega s podatki iz podatkovne baze ustvarimo in opazujemo grafe. Na sliki 3.4 vidimo primer ustvarjanja grafa, ki prikazuje število poslanih paketov po omrežju. Podatke za izris pridobimo iz podatkovne baze, v katero sta jih uvozila Node exporter in cAdvisor. Za pridobitev podatkov iz podatkovne baze uporabimo poizvedovalni jezik iz sistema Prometheus. Poizvedovalna metrika je `container_network_transmit_bytes_total`. S spremenljivkami, ki jih lahko definiramo v uporabniškem vmesniku, določimo filtriranje metrike po:

1. imenu kontejnerja: tako lahko filtriramo in vidimo število poslanih paketov samo od podatkovne baze,

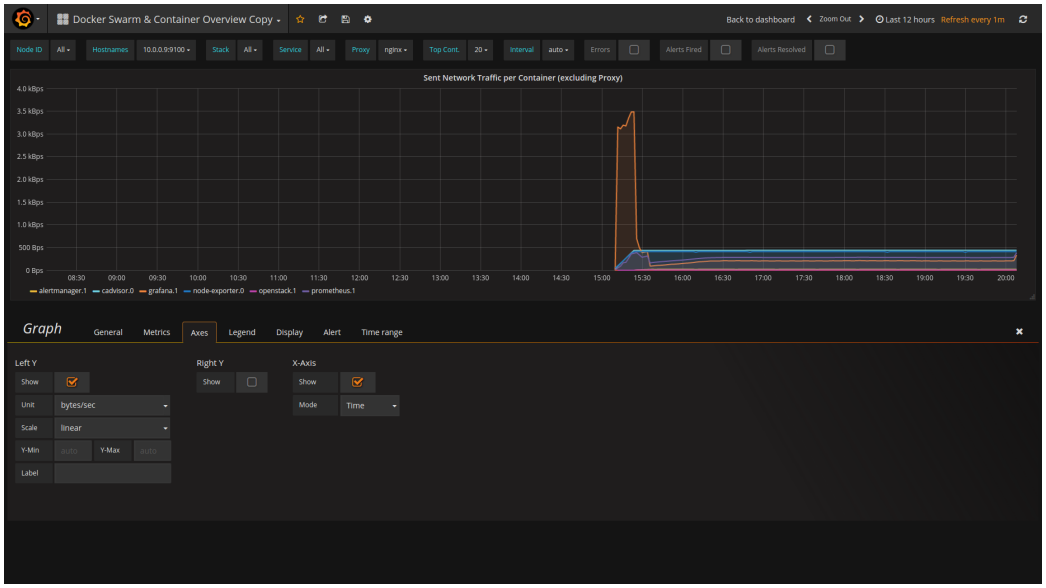
2. navideznem računalniku: lahko vidimo število poslanih paketov za specifičen navidezni računalnik.

S funkcijo `rate` pretvorimo metriko v poslane pakete na sekundo v podanem intervalu, ki je shranjen v spremenljivki `$interval`. Seštejemo vse dobljene rezultate in dobimo število bajtov glede na storitev. Dodana je tudi funkcija `topk`, ki nam omogoča, da rezultate filtriramo tako, da lahko izberemo samo prvih n storitev, ki imajo največ poslanih bajtov.

Na naslednjem zavihku nastavimo, kakšne osi bomo imeli na grafu. Primer si lahko ogledamo na sliki 3.5. Za os x izberemo časovno os, za os y pa izberemo os, ki bo prikazovala število poslanih bajtov na sekundo.



Slika 3.4: Ustvarjanje grafa.



Slika 3.5: Izbor osi na grafu.

Poglavje 4

Testni scenariji in rezultati

Testiranje spletne aplikacije smo v diplomski nalogi izvajali predvsem zato, da smo ugotovili spodnje in zgornje meje, kdaj se naša spletna aplikacija neha odzivati, oziroma kdaj ima na voljo preveč virov in je zato potrebno horizontalno skaliranje. Za testiranje smo uporabili orodje Gatling. Gatling je javanski program za izvajanje zmogljivostnih testov. Testiranje se izvaja iz našega lokalnega računalnika. Testiranje se izvaja tako, kot da bi uporabnik dejansko brskal našo spletno stran. Za zmogljivostni test potrebujemo scenarij, ki ga lahko z orodjem posnamemo ali pa ga sami napišemo v programskem jeziku java ali scala. Primer izvorne kode scenarija si lahko ogledamo na izseku izvorne kode 4.1. V kodi naštejemo, katere strani mora naš scenarij obiskati in kakšen parameter vnesti ob obisku strani. Zadnji del kode pa izvede scenarij in omeji število poslanih zahtevkov na želeno število. Orodje nam ob koncu testiranja izriše tudi grafe, s katerimi si lahko pomagamo pri ugotovitvah, kdaj se bo naša aplikacija nehala odzivati. Imamo dva testa scenarija, eden se bolj osredotoči na porabo procesorske moči, drugi pa bolj na obremenitev celotnega sistema. Med testiranjem smo opazovali grafe iz orodja Grafana in izhodne podatke testov. Ko so začeli testi dobivati napake v zahtevkih, smo pogledali grafe in zabeležili vrednost. Teste smo ponovili večkrat, da smo dobili bolj točno vrednost, kdaj začnemo dobivati napake v zahtevkih. Ko smo testiranje zaključili, smo iz dobljenih zgornjih in spodnjih

mej ustvarili alarme, na katere se bo naš sistem odzival.

4.1 Obremenitev procesorja

Obremenitev procesorja izvedemo s testnim scenarijem, ki na našo spletno stran hkrati pošlje več uporabnikov, ki se sprehodijo čez scenarij. Scenarij zajema vpis v spletni stran, obisk dveh podstrani in odjavo. Ker mora procesor procesirati veliko število zahtevkov, zelo hitro obremenimo procesor. Izvorno kodo za scenarij, ki obremeni procesor, si lahko ogledamo na izseku izvorne kode 4.1. Procesor se pri scenariju obremeni zaradi velikega števila zahtevkov na sekundo. Ko je poraba procesorja več kot 90%, so se začele pojavljati napake ob zahtevah. Zgornjo mejo smo nastavili na 70%, tako da imamo še nekaj časa, preden se začenjajo pojavljati napake. Na sliki 4.1, kjer je prikazana poraba procesorja prvega navideznega računalnika, lahko vidimo, da se, ko je procesor obremenjen na več kot 70% za več kot 5 minut, doda nov navidezni računalnik. Polovico dela torej prevzame drugi navidezni računalnik, na prvem navideznem računalniku pa se, kot je razvidno s slike, procesor razbremeni. Test se izvaja 15 minut, na spletno stran pa prožimo 14 zahtevkov na sekundo.

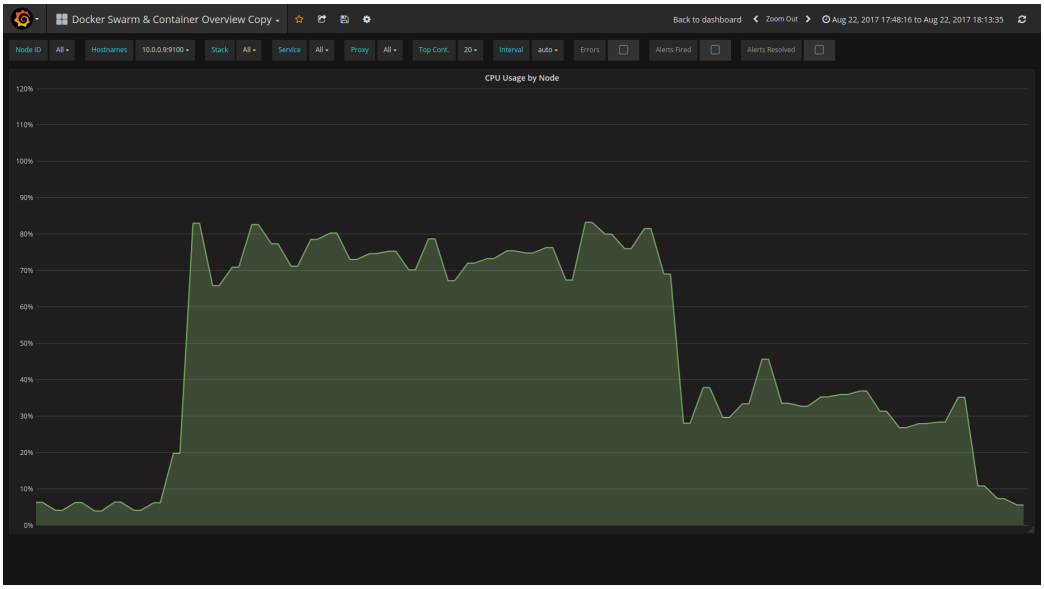
4.2 Obremenitev celotnega sistema

Obremenitev celotnega sistema (angl. *system load*) zajema scenarij, ki na naši spletni strani izvede obrazec, ki je zelo zahteven za podatkovno bazo. Obrazec poizve podatkovno bazo za 2 leti nazaj in vrne vse zapise ter nato izriše grafe. Obrazec in izrisane grafe, ki nam jih ponudi aplikacija, si lahko ogledamo na sliki 4.2. Metrika za obremenitev sistema nam pove, koliko procesov čaka v vrsti na procesiranje. Če je število procesov večje kot število procesorjev, ki jih imamo na voljo, morajo nekateri procesi čakati na procesiranje. Za obremenitev celotnega sistema imamo na voljo tri metrike:

1. `node_load1` povprečno število čakajočih procesov v zadnji minuti,

```
class CpuSimulation extends Simulation {
        val httpProtocol = http
                .baseURL("http://10.10.43.149")
                .inferHtmlResources()

        val login = exec(http("request_1")
                .post("/GCMServer/login")
                .formParam("username", "admin")
                .formParam("password", "admin"))

        val getEvents = exec(http("request_2")
                .get("/GCMServer/events"))

        val getAnalytics = exec(http("request_3")
                .get("/GCMServer/analytics"))

        val logout = exec(http("request_4")
                        .get("/GCMServer/login"))

        val scn = scenario("CpuSimulation").exec(
          ↪ login).exec(getEvents).exec(logout)
                        setUp(scn.inject(
                          ↪ constantUsersPerSec(10)
                          ↪  during(20 minutes))).
                          ↪ throttle(
                          reachRps(20) in (20
                            ↪ seconds),
                          holdFor(20 minutes)
                        ).protocols(httpProtocol)
}
```

Izsek izvorne kode 4.1: Zmogljivostni test za procesor



Slika 4.1: Graf ob bremenitvi procesorja.

2. `node_load5` povprečno število čakajočih procesov v zadnjih 5 minutah,
3. `node_load15` povprečno število čakajočih procesov v zadnjih 15 minutah.

Pri scenariju za celotno obremenitev sistema smo preko meritev ugotovili, da je metrika `node_load5` najbolj aktualna za tovrstno rešitev. Pri metriki `node_load1` je graf preveč nepredvidljiv, ker je povprečno število čakajočih procesov izračunano samo za zadnjo minuto, pri `node_load15` pa predolgo čakamo za točen graf, želimo pa hitro odzivnost, tako da nimamo preveliko napak pri zahtevah. Na sliki 4.3 vidimo, kako obremenitev sistema narašča, ko pa se ustvari nov navidezni računalnik, pa graf spet začne padati. Test se izvaja 15 minut, na spletno stran pa prožimo 3 zahtevke na sekundo.

4.3 Razbremenitev sistema

Razbremenitev sistema izvajamo tako, da ugasnemo vse testne scenarije in pustimo, da se sistem razbremeni. Ugotovili smo, da če metrika `node_load15`

ostane pod vrednostjo 0.5, lahko pobrišemo en navidezni računalnik. Proži se alarm, na katerega se bo upravljavec navideznih računalnikov odzval z brisanjem navideznega računalnika.

Naš sistem zna torej ustvarjati in brisati navidezne računalnike. Poglejmo si še, kako deluje sistem skupaj z vsemi metrikami. Na sliki 4.4 imamo graf z označenimi območji z rdečo črto. Prvo območje označuje območje, ko prvič zaženemo test. Ob prekoračitve metrike se postavi nov navidezni računalnik, ki sprosti porabo procesorja. V drugem območju še enkrat izvedemo testni scenarij, da se prepričamo, ali sistem res deluje. Nato počakamo, da se sistem razbremeni in pobriše navidezni računalnik, ko ta ni več potreben. V tretjem območju pa še enkrat izvedemo testni scenarij, ker pa je aktiven samo en navidezni računalnik, se metrika spet prekorači in je potreben zagon novega navideznega računalnika.

Slika 4.2: Obrazec in izris analitik.

Slika 4.3: Graf ob bremenitvi celotnega sistema.

Slika 4.4: Graf z vsemi metrikami.

Poglavje 5

Sklepne ugotovitve

V diplomski nalogi smo razvili sistem za avtomatsko horizontalno skaliranje spletne aplikacije. Sistem temelji na vrsti odprtokodnih programov: Docker, cAdvisor, Node-exporter, Prometheus, Grafana, Vagrant. Ko je naša spletna aplikacija preobremenjena, sistem ustrezno poveča število kontejnerjev in navideznih računalnikov, da se aplikacija razbremeni. Sistem ponuja tudi grafe o metrikah, na katerih lahko opazujemo stanje navideznih računalnikov in kontejnerjev.

Prostora za izboljšave je še veliko. Lahko bi še bolje raziskali preostale metrike in se v tem primeru odzivali na več metrik. Za avtomatsko skaliranje bi lahko uporabili umetno inteligenco, ki bi se na podlagi prejšnjih obiskov odločila, ali ustvariti nov navidezni računalnik ali ne. Recimo, da imam spletno aplikacijo, ki je najbolj uporabljena v jutranjem času. Tako bi lahko sistem naučili, da ustvari navidezni računalnik, še preden bo naša aplikacija postala obremenjena.

Menim, da smo cilj diplomske naloge dosegli, saj se naš sistem avtomatsko prilagaja glede na obremenjenost aplikacije. Tehnologije, v katerih smo razvijali, so bile za nas nove, tako da smo se v okviru diplomskega dela tudi naučili nekaj novih stvari.

Literatura

[1] Acdc. Dosegljivo: https://www.acdc-project.eu. [Dostopano: 25. 7. 2017].

[2] Alertmanager. Dosegljivo: https://prometheus.io/docs/alerting/alertmanager/. [Dostopano: 20. 7. 2017].

[3] Amazon webservices. Dosegljivo: https://aws.amazon.com/. [Dostopano: 10. 7. 2017].

[4] Ansible. Dosegljivo: https://www.ansible.com/. [Dostopano: 20. 7. 2017].

[5] cadvisor. Dosegljivo: https://github.com/google/cadvisor/. [Dostopano: 20. 7. 2017].

[6] Docker. Dosegljivo: https://www.docker.com/. [Dostopano: 10. 7. 2017].

[7] Docker swarm. Dosegljivo: https://docs.docker.com/engine/swarm/. [Dostopano: 20. 7. 2017].

[8] Google cloud. Dosegljivo: https://cloud.google.com/. [Dostopano: 10. 7. 2017].

[9] Grafana. Dosegljivo: https://grafana.com/. [Dostopano: 30. 7. 2017].

[10] Microsoft azure. Dosegljivo: https://azure.microsoft.com/. [Dostopano: 10. 7. 2017].

[11] Node exporter. Dosegljivo: https://github.com/prometheus/node_exporter. [Dostopano: 20. 7. 2017].

[12] Prometheus. Dosegljivo: https://prometheus.io/. [Dostopano: 20. 7. 2017].

[13] rkt. Dosegljivo: https://coreos.com/rkt. [Dostopano: 8. 8. 2017].

[14] Vagrant. Dosegljivo: https://www.vagrantup.com/. [Dostopano: 20. 7. 2017].

[15] Yaml. Dosegljivo: http://yaml.org/. [Dostopano: 20. 7. 2017].